Capítulo VINTE E SETE

Concorrência

Objetivos do Exame

- Usar a palavra-chave synchronized e o pacote java.util.concurrent.atomic para controlar a ordem de execução das threads.
- Usar coleções e classes de java.util.concurrent, incluindo CyclicBarrier e CopyOnWriteArrayList.

Sincronização

No último capítulo, revisamos alguns problemas que podem acontecer em um ambiente concorrente e falamos brevemente sobre bloqueio (*locking*).

Por exemplo, uma solução para evitar uma condição de corrida é garantir que apenas uma thread por vez possa acessar o código que causa o problema; um processo conhecido como sincronização desse bloco de código.

A sincronização funciona com bloqueios. Todo objeto vem com um bloqueio embutido e, como existe apenas um bloqueio por objeto, apenas uma thread pode manter esse bloqueio a qualquer momento. As outras threads não podem pegar o bloqueio até que a primeira thread o libere. Enquanto isso, elas ficam bloqueadas.

Você define um bloqueio usando a palavra-chave synchronized em um bloco ou método. Esse bloqueio é adquirido quando uma thread entra em um bloco ou método synchronized desocupado.

Em um bloco synchronized, você usa a palavra-chave synchronized seguida de uma variável de referência:

```java
Object o = new Object();

synchronized (o) { // Pega o bloqueio do objeto o
    // Código protegido pelo bloqueio
}
```

Ou a palavra-chave this:

```java
// Pega o bloqueio do objeto ao qual este código pertence
synchronized (this) {
    // Código protegido pelo bloqueio
}
```

O bloqueio é liberado quando o bloco termina.

Você também pode sincronizar um método inteiro:

```java
public synchronized void method() {
    // Código protegido pelo bloqueio
}
```

Nesse caso, o bloqueio pertence ao objeto no qual o método foi declarado e é liberado quando o método termina.

Note que sincronizar um método é equivalente a isto:

```java
public void method() {
    synchronized (this) {
        // Código protegido pelo bloqueio
    }
}
```

Código estático também pode ser sincronizado, mas em vez de usar this para adquirir o bloqueio de uma instância da classe, ele é adquirido sobre o objeto da classe que toda instância da classe tem associado:

```java
class MyClass {
    synchronized static void method() {
        /** .. */
    }
}
```

É equivalente a:

```java
class MyClass {
    static void method() {
        synchronized (MyClass.class) {
            /** ... */
        }
    }
}
```

Agora, por exemplo, se executarmos o seguinte código:

```java
public class Test {
    static int n = 0;
    static void add() {
        n++;
        System.out.println(n);
    }
    public static void main(String[] args) {
        ExecutorService service = Executors.newFixedThreadPool(4);
        Runnable r = () -> add();
        for(int i = 0; i < 4; i++) {
            service.execute(r);
        }
        service.shutdown();
    }
}
```

Ao olhar para uma possível saída, podemos ver que temos um problema de condição de corrida porque quatro threads estão executando o mesmo código:

```
2
2
3
4
```

Mas quando sincronizamos o método add() para garantir que apenas uma thread possa acessá-lo, o problema desaparece:

Classes Atômicas

O pacote java.concurrent.atomic contém classes para realizar operações atômicas, que são executadas em um único passo sem a intervenção de mais de uma thread.

Temos, por exemplo:

- AtomicBoolean, AtomicInteger, AtomicLong e AtomicReference<V>
  Para atualizar um valor do tipo correspondente (ou referência de objeto) de forma atômica.
- AtomicIntegerArray, AtomicLongArray e AtomicReferenceArray<E>
  Para atualizar os elementos do tipo de array correspondente (ou referência de objeto) de forma atômica.

Cada classe tem métodos que realizam operações como incrementos ou atualizações de maneira atômica. Veja por exemplo a classe AtomicInteger (as outras classes têm métodos semelhantes):

- int addAndGet(int delta)
  Soma atômica do valor dado ao valor atual.
- boolean compareAndSet(int expect, int update)
  Define atômica e condicionalmente o valor para o valor atualizado fornecido se o valor atual for igual ao valor esperado.
- int decrementAndGet()
  Decrementa atômica e condicionalmente o valor atual em um.
- int get()
  Obtém o valor atual.
- int getAndAdd(int delta)
  Soma atômica do valor fornecido ao valor atual e retorna o valor anterior.
- int getAndDecrement()
  Decrementa atômica e condicionalmente o valor atual em um e retorna o valor anterior.
- int getAndIncrement()
  Incrementa atômica e condicionalmente o valor atual em um e retorna o valor anterior.
- int getAndSet(int newValue)
  Define atômica e condicionalmente o valor fornecido e retorna o valor antigo.
- int incrementAndGet()
  Incrementa atômica e condicionalmente o valor atual em um e retorna o novo valor.
- void set(int newValue)
  Define para o valor fornecido.

O exemplo sobre incrementar uma variável que revisamos antes pode ser reescrito assim:

```java
public class Test {
    static AtomicInteger n = new AtomicInteger(0);
    static void add() {
        System.out.println(n.incrementAndGet());
    }
    public static void main(String[] args) {
        ...
    }
}
```

Quando executamos isso, obteremos todos os números, mas fora de ordem, porque classes atômicas apenas garantem consistência de dados:

```
1
4
2
3
```

Coleções Concorrentes

O pacote java.util.concurrent fornece algumas classes seguras para threads, equivalentes às classes de coleção de java.util.

Dessa forma, em vez de sincronizar explicitamente uma operação como esta:

```java
Map<String, Integer> map = new HashMap<>();
...
void putIfNew(String key, Integer val) {
    if(map.get(key) == null) {
        map.put(key, val);
    }
}
```

Você pode usar um ConcurrentHashMap assim:

```java
Map<String, Integer> map = new ConcurrentHashMap<>();
...
map.putIfAbsent(key, val);
```

Então, ao trabalhar em ambientes concorrentes, se você vai modificar coleções, é melhor usar as coleções de java.util.concurrent (além disso, elas frequentemente têm melhor desempenho do que a sincronização simples).

Na verdade, se você só precisa dos métodos simples get() e put() (ao trabalhar com um mapa, por exemplo), você só precisa mudar a implementação:

```java
//Map<String, Integer> map = new HashMap<>();
Map<String, Integer> map = new ConcurrentHashMap<>();
map.put("one", 1);
Integer val = map.get("one");
```

Java fornece muitas coleções concorrentes, mas podemos classificá-las pela interface que implementam:

- BlockingQueue (para filas)
- BlockingDeque (para deques)
- ConcurrentMap (para mapas)
- ConcurrentNavigableMap (para mapas navegáveis como TreeMap)
- CopyOnWriteArrayList (para listas)

Como essas interfaces estendem as interfaces de coleção de java.util, elas herdam os métodos que já conhecemos (e se comportam como uma delas), então aqui você encontrará apenas os métodos adicionais que dão suporte à concorrência.

BlockingQueue

Representa uma fila segura para threads que espera (com um tempo limite opcional) por um elemento ser inserido se a fila estiver vazia ou por um elemento ser removido se a fila estiver cheia:

	Bloqueia	**Expira**
Inserir	void put(E e)	boolean offer(E e, long timeout, TimeUnit unit)
Remover	E take()	E poll(long timeout, TimeUnit unit)

Após o tempo de espera terminar, offer() retorna false e poll() retorna null.

As principais implementações dessa interface são:

- ArrayBlockingQueue. Uma fila bloqueante limitada apoiada por um array.
- DelayQueue. Uma fila bloqueante ilimitada de elementos Delayed, na qual um elemento só pode ser retirado quando seu atraso tiver expirado.
- LinkedBlockingQueue. Uma fila bloqueante (opcionalmente limitada) baseada em nós ligados.
- LinkedTransferQueue. Uma TransferQueue ilimitada baseada em nós ligados.
- PriorityBlockingQueue. Uma fila bloqueante ilimitada que usa as mesmas regras de ordenação da classe PriorityQueue e fornece operações de bloqueio e recuperação.
- SynchronousQueue. Uma fila bloqueante em que cada operação de inserção deve esperar por uma operação de remoção correspondente por outra thread, e vice-versa. Você não pode espiar (*peek*) uma SynchronousQueue porque um elemento só está presente quando você tenta removê-lo.

BlockingDeque

Representa um deque seguro para threads (uma fila de extremidades duplas, uma fila na qual você pode inserir e remover elementos em ambas as extremidades). Ela estende BlockingQueue e Deque, e também espera (com um tempo limite opcional, após o qual retorna false ou null) por um elemento ser inserido se o deque estiver vazio ou por um elemento ser removido se o deque estiver cheio:

Primeiro Elemento (cabeça)

	Bloqueia	**Expira**
Inserir	void putFirst(E e)	boolean offerFirst(E e, long timeout, TimeUnit unit)
Remover	E takeFirst()	E pollFirst(long timeout, TimeUnit unit)

Último Elemento (cauda)

	Bloqueia	**Expira**
Inserir	void putLast(E e)	boolean offerLast(E e, long timeout, TimeUnit unit)
Remover	E takeLast()	E pollLast(long timeout, TimeUnit unit)

ConcurrentMap

Esta interface representa um mapa seguro para threads e é implementada pela classe ConcurrentHashMap.

Aqui estão alguns de seus métodos mais importantes:

- V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
  Calcula atomicamente o valor de uma chave especificada com uma BiFunction.
- V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)
  Calcula atomicamente o valor de uma chave apenas se ela **não** estiver presente no mapa.
- V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
  Calcula atomicamente o valor de uma chave apenas se ela **estiver** presente no mapa.
- V getOrDefault(Object key, V defaultValue)
  Retorna o valor da chave ou um valor padrão se a chave não estiver presente.
- V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)
  Se a chave especificada ainda **não** estiver associada a um valor (não null), associa-a ao valor dado. Caso contrário, substitui o valor com o resultado da função de remapeamento fornecida ou remove-o se for null. Isso é feito de forma atômica.
- V putIfAbsent(K key, V value)
  Se a chave não estiver presente no mapa, ela é adicionada com o valor fornecido de forma atômica.

ConcurrentNavigableMap

Representa um NavigableMap seguro para threads (como o TreeMap) e é implementado pela classe ConcurrentSkipListMap, ordenado conforme a ordenação natural de suas chaves, ou por um Comparator fornecido em seu construtor.

Um ConcurrentNavigableMap também implementa Map e ConcurrentMap, portanto possui métodos como computeIfAbsent(), getOrDefault(), etc.

CopyOnWriteArrayList

Representa uma List segura para threads, semelhante a uma ArrayList, mas quando ela é modificada (com métodos como add() ou set()), uma nova cópia do array subjacente é criada (daí o nome).

Ao iterar uma ArrayList, métodos como remove(), add() ou set() podem lançar uma java.util.ConcurrentModificationException. Com uma CopyOnWriteArrayList, essa exceção **não** é lançada porque o iterador trabalha com uma cópia **não modificada** da lista. Mas isso também significa que chamar, por exemplo, o método remove() no Iterator **não é suportado** (ele lança uma UnsupportedOperationException).

Considere este exemplo, onde o último elemento da lista é alterado em cada iteração, mas o valor original ainda é impresso dentro do iterador:

```java
List<Integer> list = new CopyOnWriteArrayList<>();
list.add(10); list.add(20); list.add(30);
Iterator<Integer> it = list.iterator();
while(it.hasNext()) {
    int i = it.next();
    System.out.print(i + " ");
    // Nenhuma exceção lançada
    list.set(list.size() -1, i * 10);
    // it.remove(); lança uma exceção
}
System.out.println(list);
```

Saída:

```
10 20 30 [10, 20, 300]
```

Com uma CopyOnWriteArrayList, não há bloqueio nas leituras, então essa operação é mais rápida. Por essa razão, CopyOnWriteArrayList é mais útil quando você tem poucas atualizações e inserções e **muitas leituras concorrentes**.

Existem outras classes como:

- ConcurrentSkipListSet, que representa um NavigableSet seguro para threads (como um TreeSet).
- CopyOnWriteArraySet, que representa um Set seguro para threads e usa internamente um objeto CopyOnWriteArrayList para todas as suas operações (então essas classes são muito semelhantes).

Além de todas essas classes, para cada tipo de coleção, a classe Collections possui métodos como os seguintes que pegam uma coleção normal e a encapsulam em uma sincronizada:

```java
static <T> Collection<T> synchronizedCollection(Collection<T> c)
static <T> List<T> synchronizedList(List<T> list)
static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)
static <T> Set<T> synchronizedSet(Set<T> s)
```

No entanto, é necessário **sincronizar essas coleções manualmente** ao percorrê-las com um Iterator ou Stream:

```java
Collection c = Collections.synchronizedCollection(myCollection);
...
synchronized (c) {
   Iterator i = c.iterator();
   ...
}
```

Além disso, elas lançam uma exceção se forem modificadas dentro de uma iteração.

Use esses métodos apenas se **você tiver que trabalhar com uma coleção existente** em um ambiente concorrente (caso contrário, desde o início, use uma coleção de java.util.concurrent).

CyclicBarrier

A palavra-chave synchronized nos ajuda a coordenar o acesso a um recurso compartilhado por múltiplas threads.

Mas isso é um trabalho de baixo nível. Quero dizer, é preciso muito esforço para coordenar tarefas concorrentes complexas. Felizmente, Java fornece classes de alto nível para implementar mais facilmente esse tipo de tarefa de sincronização.

Uma dessas classes é a CyclicBarrier. Ela fornece um ponto de sincronização (um ponto de barreira) onde uma thread pode precisar esperar até que todas as outras threads também alcancem esse ponto.

Essa classe possui dois construtores:

```java
CyclicBarrier(int threads)
```

Cria um CyclicBarrier com o número especificado de threads aguardando por ele.

```java
CyclicBarrier(int parties, Runnable barrierAction)
```

Cria um CyclicBarrier com o número especificado de threads aguardando por ele, e que executará a ação fornecida quando a barreira for alcançada.

Os métodos:

```java
int await()
   throws InterruptedException, BrokenBarrierException
```

```java
int await(long timeout, TimeUnit unit)
   throws InterruptedException,
          BrokenBarrierException,
          TimeoutException
```

Bloqueiam uma thread até que todas as outras threads tenham chamado await() (alcançado a barreira), ou, opcionalmente, até que o tempo de espera especificado termine (quando isso acontecer, será lançada uma TimeoutException).

Esses métodos lançam uma InterruptedException se a thread atual for interrompida enquanto espera, e uma BrokenBarrierException se outra thread foi interrompida ou atingiu o tempo limite, ou se a barreira foi reiniciada (com o método reset()), ou se a ação da barreira falhou devido a uma exceção.

Veja um exemplo:

```java
public class CyclicBarrierExample {
   static void checkStep(
            CyclicBarrier barrier, String step) {
      // Faz algo para preparar a etapa
      System.out.println(step + " is ready");
      try {
         // Espera as outras threads
         barrier.await();
      } catch (Exception e) { /** ... */ }
   }

   public static void main(String[] args) {
      String[] steps = {"Read the recipe",
         "Gather the ingredients",
         "Wash hands"};
      System.out.println("Preparing everything:");

      Runnable allSet = () ->
         System.out.println("Everything's ready. Let's begin.");

      ExecutorService executor =
         Executors.newFixedThreadPool(steps.length);
      CyclicBarrier barrier =
         new CyclicBarrier(steps.length, allSet);

      for(String step : steps) {
         executor.submit(
            () -> checkStep(barrier, step)
         );
      }

      executor.shutdown();
   }
}
```

A saída:

```arduino
Preparing everything:
Gather the ingredients is ready
Read the recipe is ready
Wash hands is ready
Everything's ready. Let's begin.
```

Temos três etapas e uma thread para processar cada uma. As threads irão imprimir a etapa fornecida e chamar o método await(). Quando as três threads tiverem chamado esse método, o Runnable representado pela variável allSet será executado (aquele que imprime a mensagem “Everything’s ready...” — “Tudo está pronto...”).

Como você pode ver, as etapas não são impressas em ordem, mas o programa não pode prosseguir até que todas tenham sido executadas.

Observe que o CyclicBarrier é criado com o mesmo número de threads. Tem que ser assim. Caso contrário, o programa esperará para sempre.

Se o número de threads for menor do que o esperado pelo CyclicBarrier, ele esperará para sempre pelas threads ausentes.

Para entender por que o programa ficará bloqueado quando o número de threads for maior, você precisa saber que um CyclicBarrier pode ser reutilizado e como isso funciona.

Quando todas as threads esperadas chamam await(), o número de threads aguardando no CyclicBarrier volta a zero e ele pode ser usado novamente para um novo conjunto de threads.

Dessa forma, se o CyclicBarrier espera três threads, mas há quatro, um ciclo de três será concluído e para o próximo, faltarão duas, o que bloqueará o programa.

Pontos-chave

- A sincronização funciona com bloqueios. Todo objeto vem com um bloqueio embutido e, como existe apenas um bloqueio por objeto, apenas uma thread pode manter esse bloqueio a qualquer momento.
- Você adquire um bloqueio usando a palavra-chave synchronized em um bloco ou método.
- Código estático também pode ser sincronizado, mas em vez de usar this para adquirir o bloqueio de uma instância da classe, ele é adquirido no objeto da classe que toda classe tem associado.
- O pacote java.concurrent.atomic contém classes (como AtomicInteger) para realizar operações atômicas, que são executadas em um único passo sem a intervenção de mais de uma thread.
- BlockingQueue representa uma fila segura para threads e BlockingDeque representa um deque seguro para threads. Ambas esperam (com um tempo limite opcional) por um elemento ser inserido se a fila/deque estiver vazia ou por um elemento ser removido se a fila/deque estiver cheia.
- ConcurrentMap representa um mapa seguro para threads e é implementado pela classe ConcurrentHashMap.
  ConcurrentNavigableMap representa um NavigableMap seguro para threads (como TreeMap) e é implementado pela classe ConcurrentSkipListMap.
- CopyOnWriteArrayList representa uma List segura para threads, semelhante a uma ArrayList, mas quando ela é modificada (com métodos como add() ou set()), uma nova cópia do array subjacente é criada (daí o nome).
- CyclicBarrier fornece um ponto de sincronização (um ponto de barreira) onde uma thread pode precisar esperar até que todas as outras threads alcancem esse ponto.

Autoavaliação

1. Qual das seguintes afirmações é verdadeira?

A. ConcurrentNavigableMap tem duas implementações, ConcurrentSkipListMap e ConcurrentSkipListSet
B. O método remove() de CopyOnWriteArrayList cria uma nova cópia do array subjacente no qual essa classe é baseada.
C. Um método static pode adquirir o bloqueio de uma variável de instância.
D. Construtores podem ser sincronizados.

---

2. Qual das seguintes opções incrementa corretamente um valor dentro de um mapa de forma segura para threads?

A.

```java
Map<String, Integer> map = new ConcurrentHashMap<>();
int i = map.get(key);
map.put(key, ++i);
```

B.

```java
ConcurrentMap<String, Integer> map = new ConcurrentHashMap<>();
map.put(key, map.get(key) + 1);
```

C.

```java
Map<String, Integer> map = new HashMap<>();
Map<String, Integer> map2 = Collections.synchronizedMap(map);
int i = map.get(key);
map.put(key, ++i);
```

D.

```java
Map<String, AtomicInteger> map = new ConcurrentHashMap<>();
map.get(key).incrementAndGet();
```

---

3. Dado:

```java
public class Question_27_3 {
   public static void main(String[] args) throws Exception {
      BlockingDeque<Integer> deque =
            new LinkedBlockingDeque<>();
      deque.offerLast(10, 5, TimeUnit.SECONDS);
      System.out.print(
         deque.pollLast(5, TimeUnit.SECONDS)
         + " "
      );
      System.out.print(
         deque.pollFirst(5, TimeUnit.SECONDS));
   }
}
```

Qual é o resultado?

A. O programa apenas imprime 10
B. O programa imprime 10 e trava para sempre
C. Após 5 segundos, o programa imprime 10 e, após mais 5 segundos, imprime null
D. O programa imprime 10 e, 5 segundos depois, imprime null

---

4. Sob quais circunstâncias o método await() de CyclicBarrier pode lançar uma exceção?

A. Se a thread for dormir
B. Quando a última thread chamar await()
C. Se alguma thread for interrompida
D. Se o número de threads que chamam await() for diferente do número de threads com o qual o CyclicBarrier foi criado.